Welcome to Lab Memory!

Course Website: https://courses.engr.illinois.edu/cs225/sp2022

Overview

In this week's lab, you will learn about memory management: how to allocate and de-allocate memory correctly in your program. You will discover ways of memory management, and practice spotting memory bugs in the code.

Destructors

Destructors (dtors) are special member functions of classes. They are the opposite of constructors: their job is to release (de-allocate) memory when an object of the class is no longer needed. Destructors are automatically called when an object runs out of its scope; we never explicitly call a destructor, instead it is implicitly called when we use the keyword “delete” or when the lifetime of the object ends.

Exercise 1.1: Write the destructor for the Orchard class.

Exercise 1.2: On which line(s) will the destructors be called in main.cpp (see next page)? ~Orchard: _12__ ~Tree: _12, 13, 14_

---

### orchard.h

```cpp
#pragma once

class Tree {
public:
    string fruitName;
    double yield;
};

class Orchard {
public:
    static const int MAX_TREES = 500;
    Orchard();
    bool addTree(Tree& t);
    Orchard(const Orchard& other);
    Orchard& operator = (Orchard const & other);
    ~Orchard();
private:
    int size_;
    Tree* trees_;
};
```

---

### orchard.cpp

```cpp
#include "orchard.h"

Orchard::Orchard() {
    size_ = 0;
    trees_ = new Tree[MAX_TREES];
}

bool Orchard::addTree(Tree& t) {
    if (size_ < MAX_TREES) {
        trees_[size_] = t;
        size_++;
        return true;
    } else {
        return false;
    }
}

Orchard::Orchard(const Orchard& other) {
    trees_ = new Tree[MAX_TREES];
    for (int i=0; i<other.size_; i++) {
        trees_[i] = other.trees_[i];
    }
    size_ = other.size_;
}

Orchard& Orchard::operator = (Orchard const & other) {
    if(this != &other){
        for (int i=0; i<other.size_; i++) {
            this->trees_[i] = other.trees_[i];
        }
        this->size_ = other.size_;
    }
    return *this;
}

Orchard::~Orchard() {
    delete[] trees_;
    trees_ = null;
}
```
Memory Errors

Memory errors occur when memory access is mismanaged; some ways it can occur are through: 1) invalid memory access in heap or stack, 2) mismatched allocation/deallocation, or 3) missing allocation or uninitialized variable access (eg. dereferencing NULLs). Memory errors often result in “segfaults” when the program is run.

**Exercise 2.1:** What will line 7 in `main.cpp` print out? ___012345678910___(possibly)segmentation fault____

**Exercise 2.2:** A memory error will occur somewhere between lines 10 and 16. Find and correct this error.

Memory Leaks

*Memory leak* is a type of Memory Error. Memory leaks most commonly occur when heap memory is no longer needed but is not correctly released (*still reachable block*), or when and object/variable is stored in memory but cannot be accessed by the running code (*lost block*). Memory leaks are often harder to detect than memory errors as they won’t always cause an error at runtime. Debugging tools such as Valgrind can help detect memory leaks.

**Exercise 3:** For each memory block allocated in the code below, decide if it has been released correctly. If not, add code to correctly release it.

In the programming part of this lab, you will:
- Learn about one memory debugging tool: Valgrind
- Complete the given code for lab_memory
- Debug the given code by correcting memory errors and memory leaks

*As your TA and CAs, we’re here to help with your programming for the rest of this lab section!* 😊